Using the Scalable Parallel Random Number Generator (SPRNG) Library in the Milagro Implicit Monte Carlo (IMC) Code

LA-UR-02-0624

Todd J. Urbatsch
Thomas M. Evans

CCS–4, Transport Methods Group
Los Alamos National Laboratory
Los Alamos, NM 87544

SPRNG Workshop
Sandia National Laboratory
February 11-12, 2002

Abstract. We describe briefly the Milagro Implicit Monte Carlo (IMC) Code for nonlinear thermal radiative transfer calculations. In particular, we describe how Milagro uses the Scalable Parallel Random Number Generator (SPRNG) Library for its random number generation. The encapsulation of the SPRNG Library has fit nicely into Milagro's levelized design and component tests. We have enjoyed success with Milagro and SPRNG. We conclude with a few issues that we would like the SPRNG developers to address.

Outline

1. The Milagro Implicit Monte Carlo (IMC) Code
2. SPRNG in Milagro
3. Success with Milagro and SPRNG
4. SPRNG Usage Issues

Milagro IMC

Fleck & Cummings Implicit Monte Carlo method for non-linear thermal radiative transfer:

- used for astrophysics, inertial confinement fusion (ICF)
- nonlinear: material and radiation codependent
- linearized over each timestep
- time-implicitness: absorption/reemission represented as an effective scatter

The Milagro IMC Code:

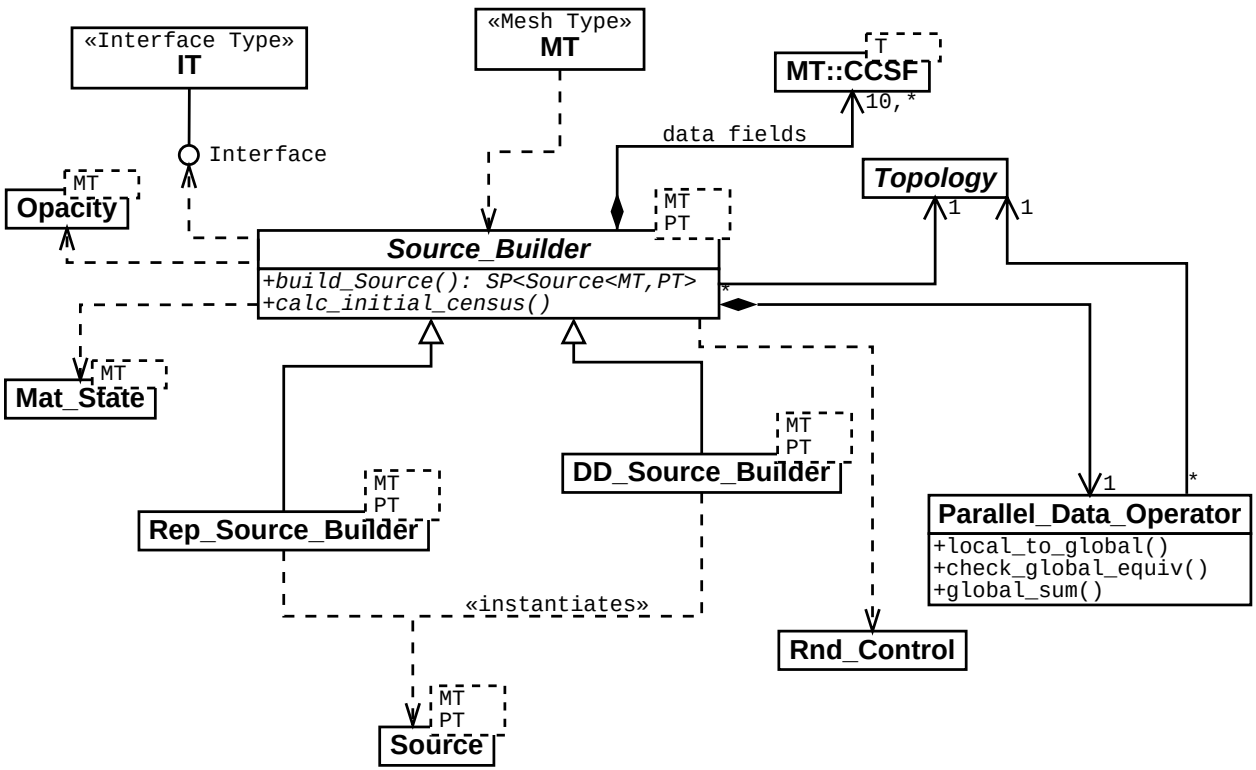
- levelized design
- object oriented C++
- templated on mesh type
- testing
  - Design-by-Contract™
  - component testing
  - regression testing
  - verification problems
- parallel

Milagro Design Concepts

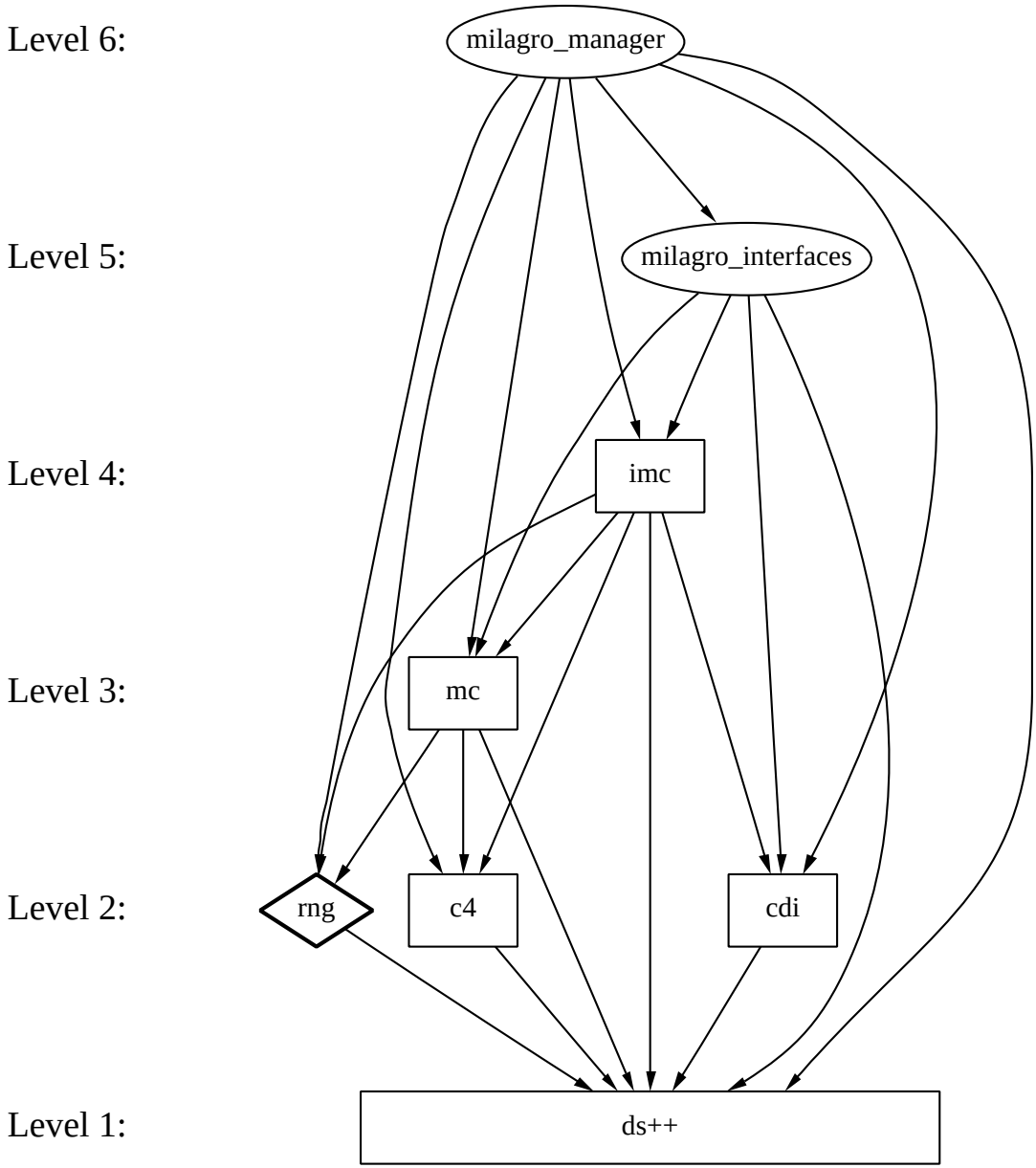
- written in C++ using Object-Oriented/generic design concepts;
- design is levelized both physically and logically;
- testing in 3 levels: component, code, regression;
- Design-By-Contract is used for run-time verification;
- major components are templated on the discretized independent parameters: space and frequency;
- variations of the *Factory Method* and *Builder* pattern are used to separate disparate input conditions from the objects that perform the transport calculation;
- code is designed into components that can be reused for different packages;
- the same subset of components have been used to build three distinct packages: **Milagro**, **Wedgehog**, and **Milstone**;
- we use the Common Data Interface (**cdi**) to access data through a common interface specification;
- multiple parallel topologies are easily supported through the application of the *Factory Method* pattern.

Factory Pattern for Source

Two generalizations of the `Source_Builder` are used to build the source on each processor depending on the desired parallel topology. The `Source_Builder` does load-balancing calculations and ensures that the problem is reproducible, regardless of the topology.

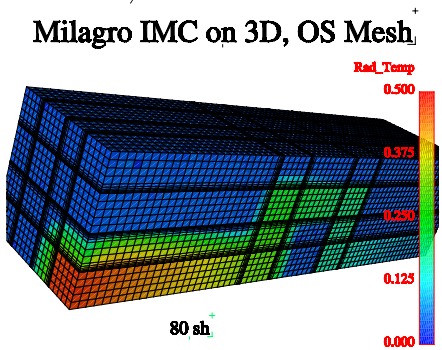


Milagro Levelization Diagram



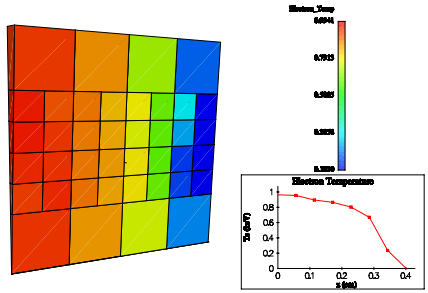
Milagro's Mesh Types

- Orthogonal Structured, Nonuniform

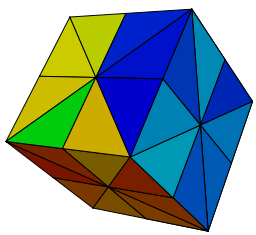


- Orthogonal AMR

- RZWedge AMR

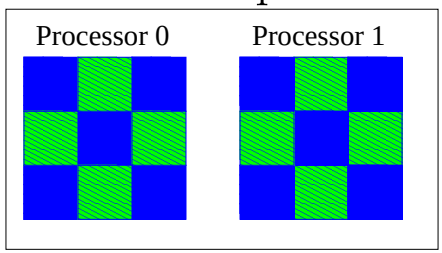


- Tetrahedral (not integrated yet)

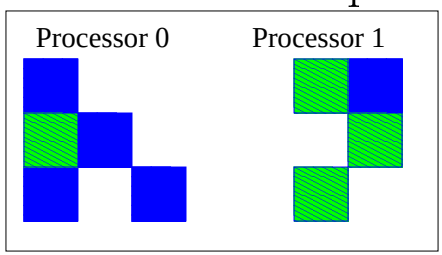


Milagro's Parallel IMC

- Reproducible
- Distributed Memory Model
  1. Domain Replication



  2. Domain Decomposition



- Future: Shared and Mixed Memory Models
  1. OpenMP
  2. OpenMPI

Reproducibility in IMC

- Reproducible IMC $\Rightarrow$ Reproducible Particles
  $\rightsquigarrow$ Give each particle its own RN generator

- Reproducible Particles $\Rightarrow$ Reproducible generator IDs
  $\rightsquigarrow$ Construct two arrays per cell per processor:
  1. starting generator IDs
  2. number of particles to run

Reproducibility on a Replicated Mesh

Example: 2 cells, 3 particles per cell, 2 processors

	global	processor 0	processor 1
# particles	3 3	2 1	1 2
starting ID	0 3	0 3	2 4

- Number of Particles
  1. divide particles evenly among the processors
  2. leftover particles go to a sliding string of processors
  3. requires no communication

- Starting ID
  1. maintain running cell-dependent offset (global)
  2. loop over cells, summing the following
     (a) global offset
     (b) even-split particles from prior processors
     (c) leftover particles from prior processors
  3. requires no communication

Reproducibility on a Decomposed Mesh

Example: 2 cells, 3 particles per cell, 2 processors

	global	processor 0	processor 1
# particles	3 3	3 3	3 3
starting ID	0 3	0 3	0 3

- Starting ID
  1. requires global mesh-sized array of number of particles
  2. requires global mesh-sized array of starting random numbers
  3. requires global collapses of data

SPRNG Wrappers

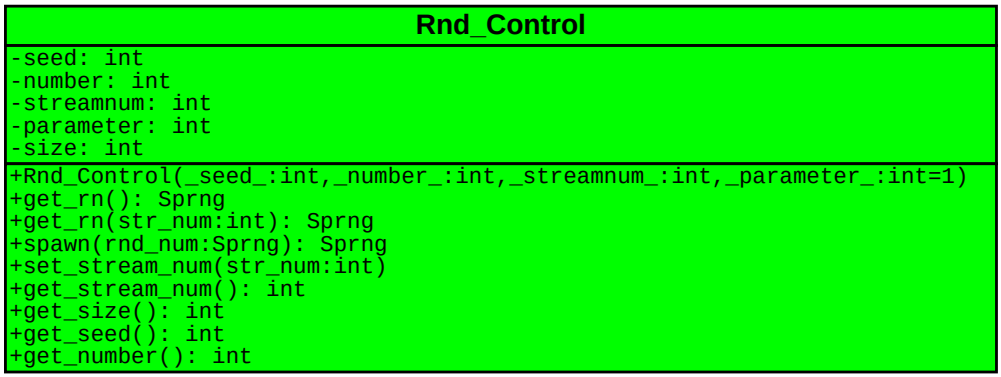
The SPRNG random number functionality was abstracted in our packages through a set of C++ wrappers. The wrapper design is a simplified variant of the *Factory Method* pattern [Gamma et al. 1995]. The wrappers meet the following requirements.

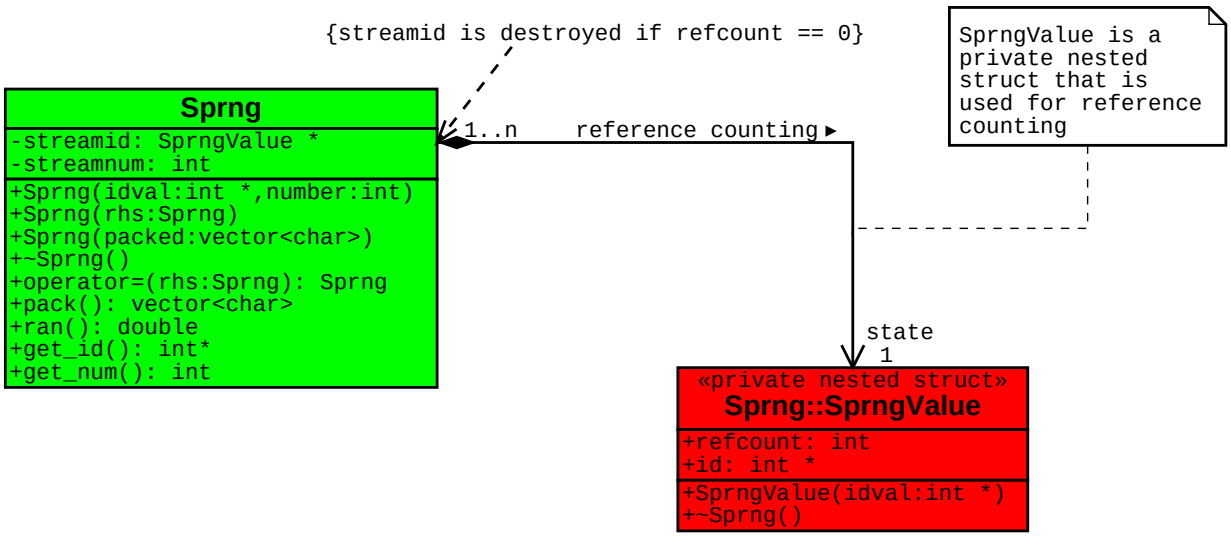
- Hides SPRNG implementation (function calls) from clients;
- Protects memory implementation of SPRNG; it is memory-safe;
- Creates independent random number generators that are specified by user-input stream numbers;

The **rng** package contains the wrapper classes we use to access SPRNG.

RNG Package

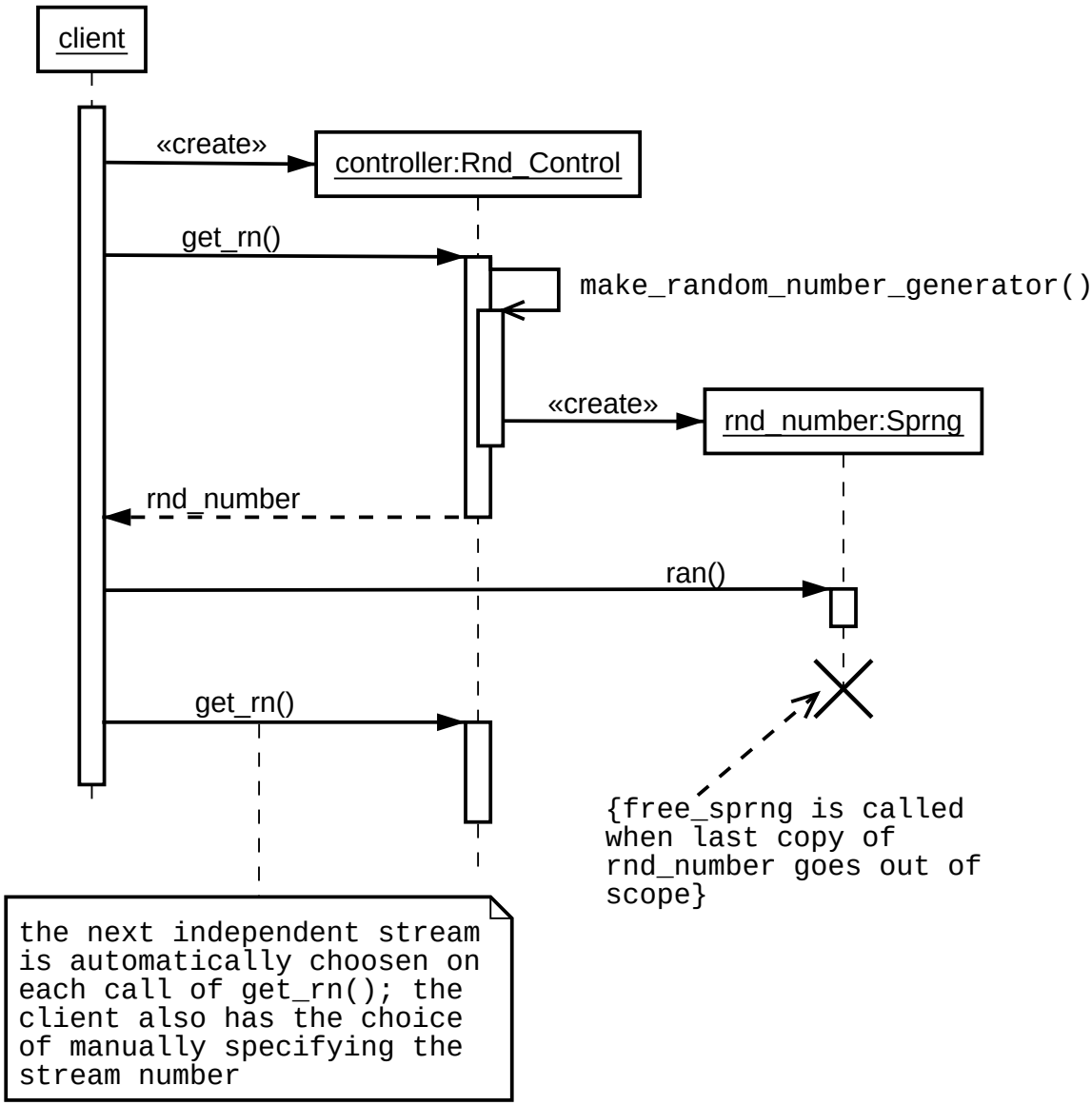
The package consists of two classes (plus tests): `Rnd_Control` and `Sprng`.





RNG Usage

Usage is straightforward:



Success with SPRNG

- SPRNG fits into our object-oriented, levelized design
  - SPRNG encapsulates random number generation
    - theory
    - expertise
    - algorithms
    - coding
    - testing
  - RNG: our fully-tested wrapper
    - allows for different SPRNG generators
    - allows for different vendors besides SPRNG
- Milagro has been successful without RNG worries
  - passed verification tests
  - design/development practices have allowed confident refactoring
  - rigorous testing yeilds an IMC capability that actually works for the users, right out of the box
  - matched experimental ICF results

SPRNG Usage Issues

1. Access limited to size_of_int=2^{32} generators

2. Different random number generators for different initial number of generators

3. Size of states

Issue 1: Limited Access

Access limited to size_of_int=2^{32} generators

1. Defeats the whole purpose of using SPRNG's generators with a gazillion different random number streams

2. Fixes
   (a) wrap generator ID around back to zero
      (i) works now and is in use
      (ii) incurs modulo cost
      (iii) wrap around at 1e9 (that's the value of num_gens)
   (b) make num_gens a "long long"
      (i) system must treat long-longs as 8-byte integers
      (ii) C++ standard doesn't specify
   (c) make num_gens an array

3. Questions and Discussion
   (a) can num_gens be made a long-long?
   (b) current workaround may be adequate...
   (c) iff, wrapping around at num_gens is okay.
   (d) ideally num_gens should be an 8-byte integer

Issue 2: Effects of Changing Initial num_gens

Different initial num_gens gives different streams

1. Initially, we wrapped around at 2e9 generators

2. Realized that we had only initialized 1e9 generators
   $\rightarrow$ oops, error!

3. Different streams with new num_gens

4. Spawning causes this nonreproducibility

5. Questions and Discussion
   (a) can the applicable generators be made reproducible independent of initial num_gens?
   (b) should they?
   (c) for IMC, spawning isn't heavily taxed

Issue 3: RN State Size

Large memory cost for storing state with each particle

1. We store the RN state with the census particles

2. Memory isn't a problem yet, but it may be soon

3. Potential workaround for state sizes
   (a) use smaller lags in Fibonacci
   (b) use different generator
   (c) don't store, recreate RN state of reborn census particles
      - brute force: store generator ID and number of random numbers used, then spin the dial
      - analytic way to jump ahead – do all generators have this capability?

4. Questions and Discussion
   (a) what's the status of smaller lags with SPRNG?
   (b) our high-level regression tests are tied to the generator (lots of work to change generator types)
   (c) Can the RN state be recreated instead of stored?
      - takes up 2/3 of a census particle's storage
      - can it be done efficiently?

Conclusion

We have been very happy with SPRNG. Its encapsulated, high quality capability has fit nicely into our Monte Carlo code development. We look forward to SPRNG addressing our current existing issues, and we hope that our use of SPRNG will continue to influence and benefit its maintenance, research, and development.